Midterm Exam

CPT S 411: Introduction to Parallel Computing

Fall 2020

Exam given (on Blackboard): 11:00am PT, December 1, 2020
Exam due (on Blackboard): 11:59am PT, December 3, 2020
(i.e., it is due one minute before noon on Thursday. No email submissions allowed.)

Name: Nate Jensvold

ID#: 11588828

Special Instructions:

- Make sure your exam copy has 10 sheets (including this cover page).
- Fill both your name and student ID. Exams without these fields will not be graded.
- This is a Take Home exam. You are welcome to refer to any resource available on the course' webpage (including lecture notes and instructor scribes). However, you should NOT search online or refer to any other external link or consult with anybody (either in class or outside). 100% of the solutions provided should be yours. If there is any evidence to suggest otherwise, it will be treated as cheating, and the test will be scored with a direct F grade and the student(s) involved will be reported to the university's office of student conduct and academic integrity. There will be NO negotiation on this aspect. So please take this seriously.
- Answer each question within its designated blank space. If you need more space, use the backside of the corresponding page.
- When asked for run-time complexity (for MPI/distributed memory), express them in two parts: computation time + communication time, using the notation we have always used in class.
- Use of figures to illustrate your idea(s) is encouraged wherever possible. Figures can save a lot of text!
- Unless otherwise stated, we will follow the notation that we have been using throughout the course: n for input size, p for number of processors, and other conventions we have been following throughout the course.

Problem	1	2	3	4	5	6	7	8	Bonus ques.	**Total+bonus**
Points	12	7	6	5	8	10	6	8	(+10)	**62 (+10)**
Your Score										

1. (12 points) **Fundamental concepts:**

i) A parallel program, when run on 4 processors, completes in 20 seconds; and when run on 1 processor completes in 60 seconds. What is the relative speedup and efficiency at 4 processors?

S=T(p=1)/T(p=x)
S=60/20
Speedup = 3.0
E=S/p
E=3.0/4
Efficiency=75%

ii) For a given problem, the serial work for the best performing serial code is $\Theta(\mathbf{n^2})$, and a parallel algorithm has a runtime complexity $\Theta(\frac{n^2}{p} + p \lg p)$. Write the corresponding expressions for the algorithm's: a) parallel work complexity, and b) parallel overhead.

Parallel work complexity = $\Theta(p\left(\frac{n^2}{p} + p \lg p\right))$
Parallel overhead = $T_o(n,p) = p \times T(n,p) - \omega$
Parallel overhead = $p \times T(n,p) - \Theta(n^2)$
Parallel overhead = $\Theta\left(\frac{n^2}{p} + p \lg p\right) - \Theta(\mathbf{n2}) = \mathbf{\Theta}\left(\frac{\mathbf{n^2}}{\mathbf{p}} + \mathbf{p \lg p} - \mathbf{n^2}\right)$

iii) Check <u>all</u> that apply (no need for justification):

Algorithm *A* gives better speedup than algorithm *B* when run on *p* processors. This implies that:

a) *A* has more efficiency than *B* on *p* processors. √

b) *B* has more efficiency than *A* on *p* processors.

c) *A* is guaranteed to have more efficiency than B on p/2 processors.

d) *A* and *B* can possibly have the same efficiency on p/2 processors. √

e) *A* and *B* can possibly have the same efficiency on 2*p* processors.

f) none of the above

iv) Choose the correct answer (justify your choice with a brief explanation):

If you assign 100 processors to an input that contains 1000 elements, how many processors will you need to process a new input that contains 2000 elements in roughly the same amount of time?

a) 50

b) 200

c) 400

d) none of the above

e) need more information to answer the question

You will need more information to answer this question because we know nothing about how the algorithm scales as more processors are added. There's the possibility that the algorithm could have the communication overhead increase at a non linear rate meaning more than 2x the processors would need to be used for an input size of 2000

2. (2+2+3 = 7 points) **Fundamental concepts:**

For a parallel code that I have written and tested, the following table summarizes the run-time results (in seconds):

	Number of processors p					
n	**1**	**2**	**4**	**8**	**16**	**32**
64	1,000	499	255	120	85	90
128	4,010	2,000	999	510	255	175
256	16,100	8,000	3,980	2,010	1,020	590
512	64,020	32,000	15,800	8,020	4,020	2,200

Based on this table, and without any further information on the underlying algorithm or the problem, answer the following questions:

(a) What would be your best estimate of the code's serial work complexity (i.e., ω)?
$\omega = 4n$

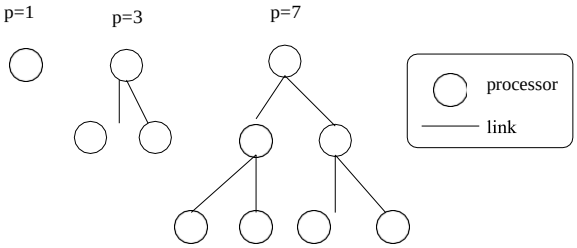
(b) What is the relative speedup of the parallel code on 16 processors for n = 256?
RS = T(n,1)/T(n,p)
RS = 16100/1020 = **15.78**

(c) If the scaling trend continues, then roughly how much time do you think the program will take to complete, on n = 1024 using p = 64?

2200*4 / 2 = **4400**

Note: If you are unable to calculate these figures without a calculator then show me the final expressions (or your best rough estimates).

3. (6 points) **Network topology:** Consider a new kind of network topology where the processors are arranged as a perfectly balanced complete binary tree. We will assume here that the number of processors is always one less than a power of two — i.e., $p = 2^k - 1$, for $k > 0$. Examples of such a network are shown below for small values of p.



Calculate the network parameters for this network topology: i) network diameter, ii) number of links per node (if variable, give the range), and iii) network bisection bandwidth. Your answers should be expressed as a function of p and k as appropriate. Use of asymptotic notation is allowed, wherever you can't be exact.

network diameter = $(k-1)^2$

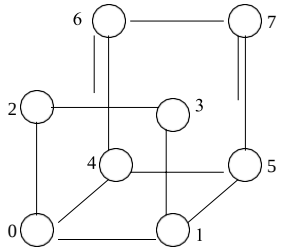
number of links = 3 (interior nodes are 3 while children and root are 1)

network bisection bandwidth = 1 (only need to chop the top node to split graph)

4. Network topology:

a) (5 points) On a hypercubic network with p processors, we want to compute the shortest path distance (in hops) between any two processor ranks i and j. Provide a simple and efficient algorithmic pseudocode to compute the shortest path distance between any two arbitrary ranks i and j. For example, in the 8-processor hypercube shown below, the shortest path distance between ranks 1 and 6 is 3, while the distance between ranks 1 and 7 is 2. Note that there could be more than one shortest path between any two processors.

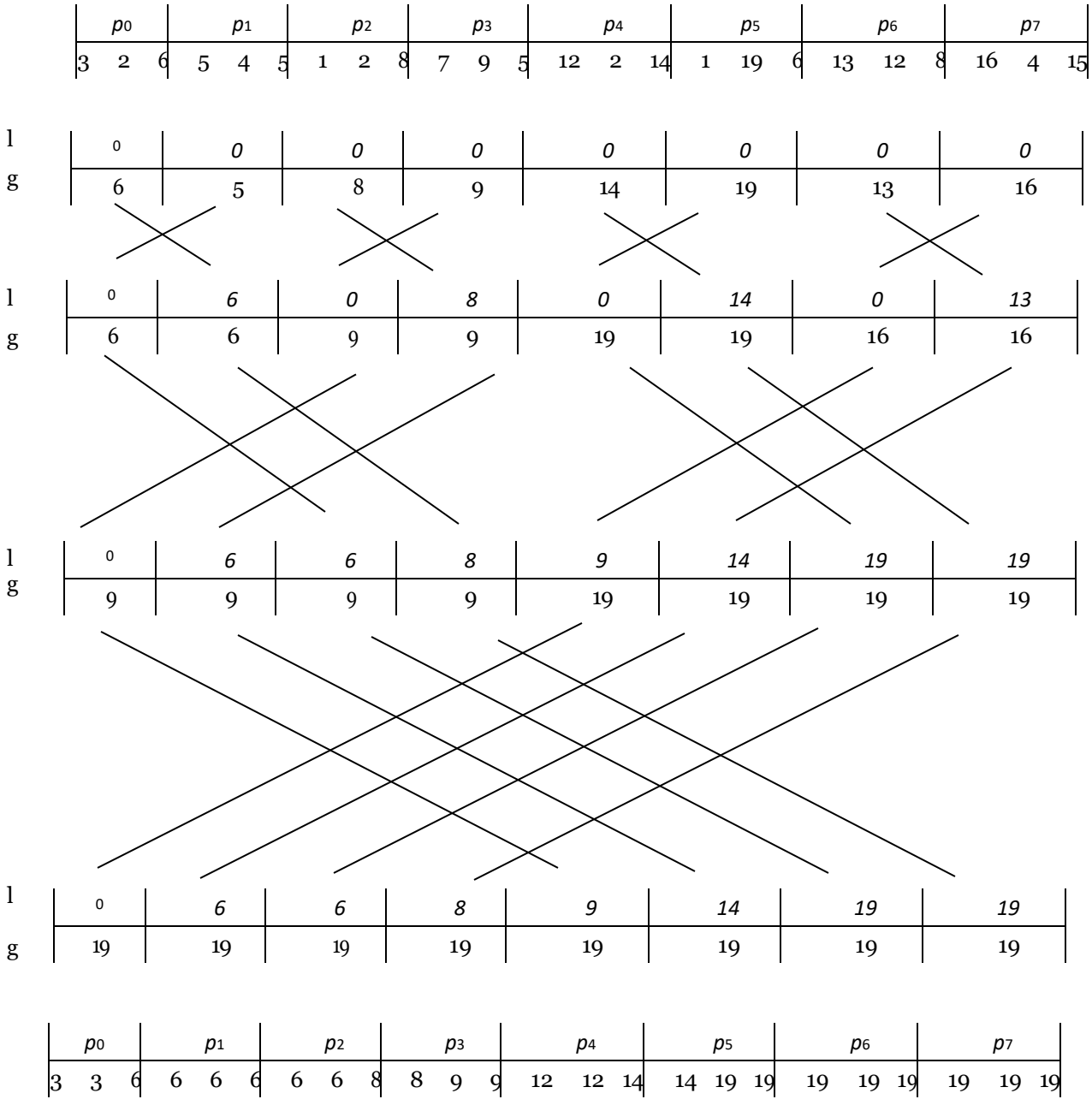
If you express the main idea behind your algorithm that should suffice. Obviously your algorithm should be generic (not specific to any example).




```
def BFS(G, I, j)
        # G is the cube, i in the starting node
        Q = queue()
        q.enqueue(s)
        s.visited = True
        while (len(q) > 0)
                v = Q.dequeue()
                for all neighbors e of v:
                        if  e .visited == false:
                                q.enqueue(e)
                                return v.distance
                                # v.distance is the number of parents/grandparents between I and j
                v.visted = True
```

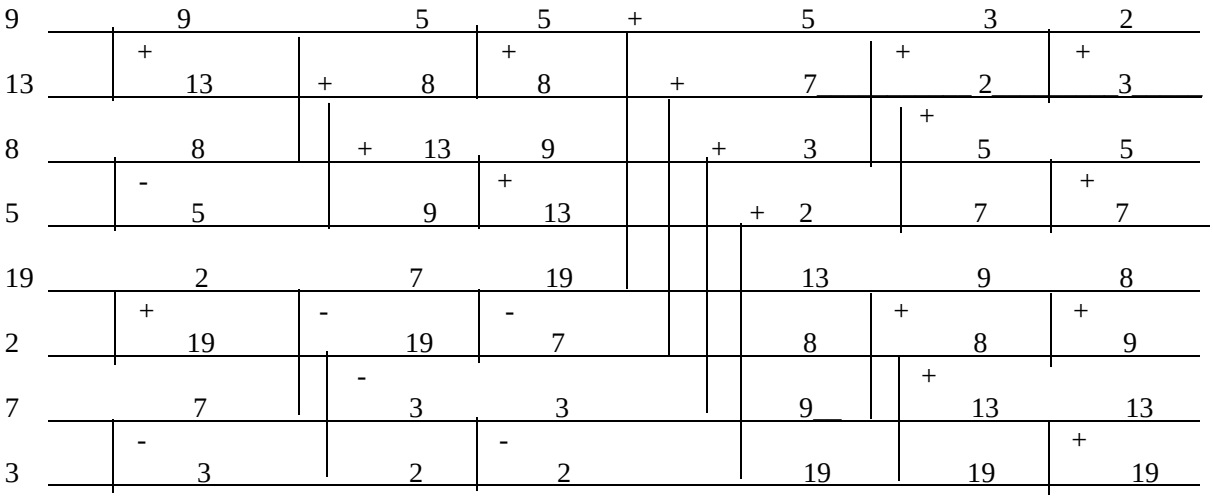
This algorithm will perform breadth first search on each node it comes across. Once j has been found it calculates the number of nodes between I and j by looking at the number of parents/grandparents between the two nodes.

5. (8 points) Simulate the parallel prefix algorithm to compute the **prefix maximum** of the following 24-element array on 8 processors, by showing the values generated at all intermediate steps and at output. Note that for prefix maximum, your i^{th} output value should be the cumulative max of all values from index 0 to i of the global array. For instance, the 0^{th} output value will be output by p_0 and it will be 3, while the 7^{th} output value will be output by p_2 and will be 6. Your solution should show all the intermediate steps in going from this input to the corresponding output using parallel prefix (consistent with the examples used during class lectures).

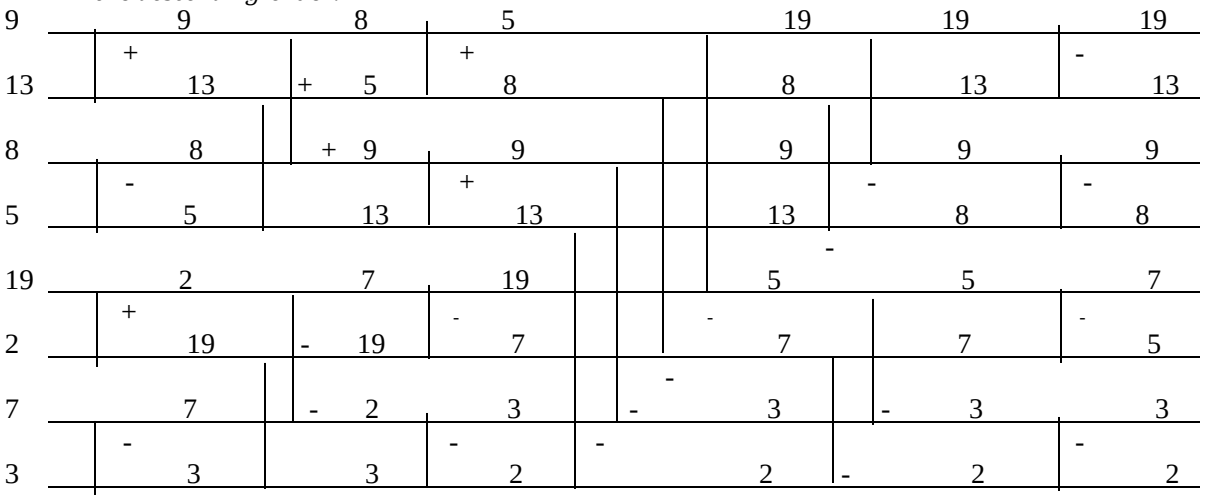


6. (10 points) **Sorting:**

i) Use bitonic to sort the below set of numbers in the *ascending* order. Your solution should show the bitonic circuit (communication steps), consistent in the manner we discussed in class, leading from the input (left) to the output (right).



ii) Now, redo the above exercise above but with the goal of directly bitonic sorting the numbers in the *descending* order.



7. (6 points) **OpenMP programming:** Write an OpenMP multithreaded program for counting the **number of distinct columns** of an $n \times n$ matrix. A column is *not* distinct if it has at least one another column that is an identical "twin" (i.e., copy) in the matrix. Assume that $n > p$ where p is the number of threads.

   Show only the main parallel part of your code (i.e., #pragma omp parallel region). Full syntactic correctness is not necessary but I will be looking for the main logic behind your code and what implementation choices you make regarding:

   (a) the variable scoping decisions you made for each variable used;

   (b) the scheduling decision you made for any for loop; and

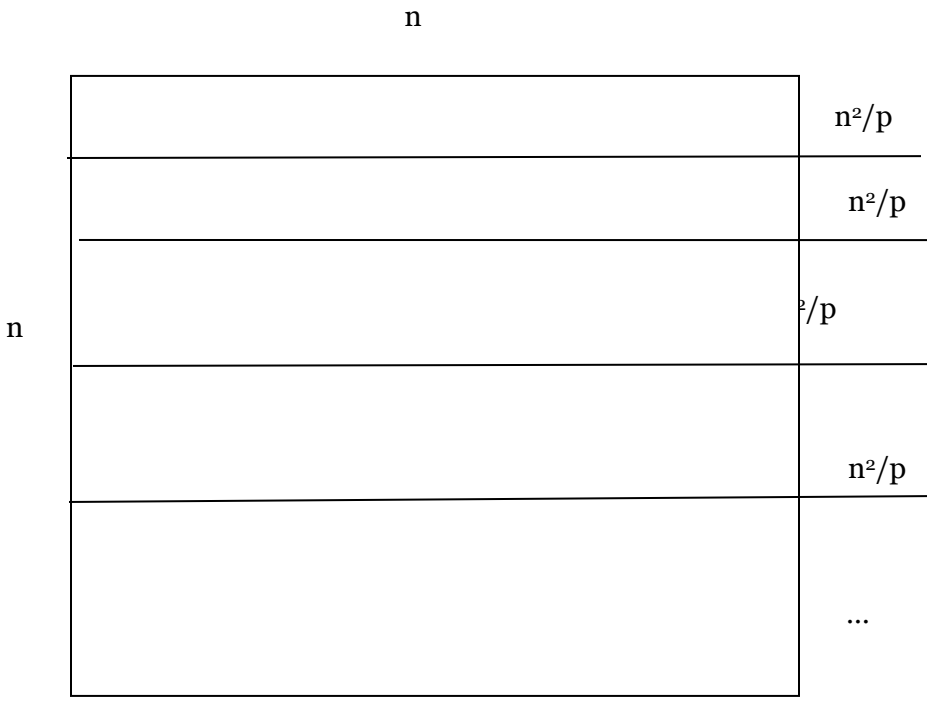
   (c) the use of any synchronization primitives if applicable.

```
Int matrix = …
Int uniqueColumns = 0
Int n = atoi(argv[1]);
Int p = atoi(argv[2]);
omp_set_num_threads(p);
int columnStart = 0;
int rank = 0;
#pragma omp parallel private(columnStart, rank) reduction(+uniqueColumns)
{
        Rank = omp_get_thread_num();
        ColumnStart = n/p * rank;
        for (int i=0; i< n/p;i++)
        {
                if distinctColumn(matrix[i+columnStart]):
                {
                        uniqueColumns++;
                }
        }
}
```

8. Distributed memory parallel algorithm (8 points):

Given an $n \times n$ binary matrix **M** (i.e., all values are either 0 or 1), a row i is called "invariant" if *all* values in that row are either <u>all 0s</u> or <u>all 1s</u>. (i.e., Even if one cell has a different value in that row becomes variant.)

Assuming that the matrix is too large to fit in a single process memory and assuming that each process can afford only to store only $O(n^2/p)$ part of the matrix, give a distributed memory parallel algorithm to *count the number of invariant rows* of the matrix **M**. For this part, it is better you draw a figure first, and show how you plan to partition the matrix across the p processes, before giving the pseudocode (no need for any real code) to show what each process will do. I will be looking for the main idea behind your approach (i.e., how you partition the input, steps which are computation, and steps which are communication, etc.).




```
int isInvariant(int arr[], int n)
{
        int zresult = 0;
        int oneResult = 0;
        for (int i=0; i<n; i++)
                if (0 == arr[i])
                        zresult ++;
                elif (1==arr[i])
                        oneResult++;
        if zresult == n or oneResult == n:
                return True
```

```
        else return False
}


Int M = …
Int n = atoi(argv[1]);
Int p = atoi(argv[2]);
omp_set_num_threads(p);
int startingrow = 0;
int rank = 0;
invariantRows = 0;
#pragma omp parallel private(startingrow, rank) reduction(+invariantRows)
{
        Rank = omp_get_thread_num();
        startingRow = n/p * rank;
        for (int i=0; i< n/p;i++)
        {
                if isInvariant (matrix[i+columnStart], n):
                {
                        invariantRows++;
                }
        }
}
```

9. **A "Cold Wintry" Bonus question** (10 points)

On a snowy day, you need to remove snow from your driveway so that you can drive your car out of the garage. There is already n cubic feet of snow. If you shovel alone then the snow can be removed at the rate of x cubic feet per second. You can ask for help and every new person added to the task will be able to remove at that same rate. But snow is continuing to fall at y cubic foot per second, where $y < x$. You can drive your car only when the driveway becomes empty (at least momentarily). The questions for this problem are as follows:

a) What are the serial and parallel times ($T(n, 1)$ and $T(n, p)$ respectively) for shoveling before you can drive your car off the garage?

T(n,1) = n/(x ft³/s – y ft³/s)t

T(n,p) = n/(p(x ft³/s) – y ft³/s)t

b) What is the speedup achieved by p people (including yourself)?
S = T(n, 1)/T(n,p)

S = (n/(x ft³/s – y ft³/s)t) / (n/(p(x ft³/s) – y ft³/s)t)

$$S = \frac{n\,ft^3}{(x\frac{ft^3}{s} - y\frac{ft^3}{s})} * \frac{\left(px\frac{ft^3}{s} - y\frac{ft^3}{s}\right)}{n\ ft^3} = \frac{\left(px\frac{ft^3}{s} - y\frac{ft^3}{s}\right)}{\left(x\frac{ft^3}{s} - y\frac{ft^3}{s}\right)} = \frac{px - y}{x - y}$$

S $= \frac{px-y}{x-y}$

c) Can you comment on the achieved speedup — i.e., is it linear, super-linear or sub-linear?
The achieved speedup is linear because of the constant rate that the snow is falling. The snow will continue to fall at rate y no matter how many people decide to assist with the removal of the snow from the driveway. This means that as more and more people join in on the snow removal it will continue to grow and scale faster than the snow falling ie(the ratio between p=1 and p>=1 will grow at a fixed rate equivalent to the growth of p).